Floating Platform Gravitational Adventure

Group 2
Link to watch demo

March 2025

1 Introduction

Floating Platform Gravitational Adventure is a multiplayer platformer game. The goal is to jump up randomly generated platforms quickly to be the first one to claim the coin at the top. An FPGA is used as the controller to move left, move right, jump and dash. The game state is synced using an AWS EC2 server and data is stored using a DyanomoDB database. The game can be played by two different people, each using their own laptop, being connected over AWS.

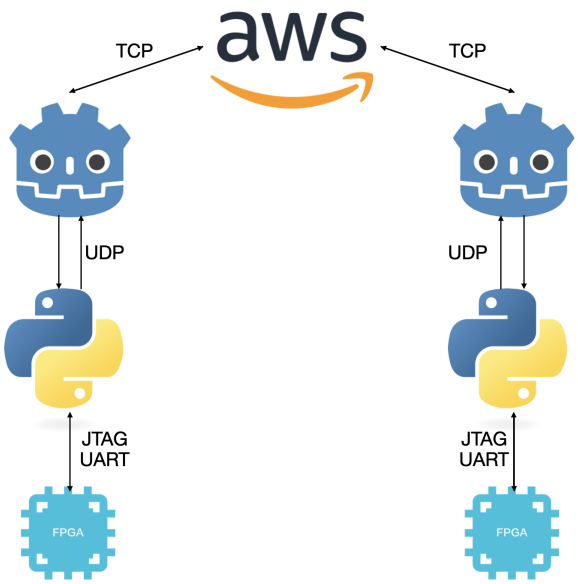

Figure 1: System Architecture

The game first connects to an AWS server, which generates the platform positions. When a client joins, it synchronizes with the server, ensuring that all players see the platforms in the same locations. Controller data is generated, parsed and sent from the FPGA to a local Python server over UART. The server then forwards this data via UDP to the game engine, where the data is decoded to determine the player's actions, such as jumping, moving left, or dashing. This player motion also integrates beyond left and right, the amount the FPGA is tilted is proportional to the speed in game. By flicking the FPGA, the character receives a temporary horizontal speed boost. The game begins with a start screen, where players select their character. They are then placed in a waiting screen until their opponent selects their character. Once both players have chosen, they are loaded into the game and compete on the same randomly generated map. The position of each player is sent to the AWS server each frame, and this data is read by the other player's client. This ensures that both players are positioned correctly on each screen, enabling the multiplayer system.

2 Functionality

2.1 FPGA

The hardware is designed to capture and process the movements and actions on the FPGA, then pass this information to the game engine for interpretation. Additionally, it retrieves the current live score and displays it for the user. The DE10-Lite FPGA development kit serves as the game controller, while Platform Designer, Eclipse, and a Python server allow for communication and data transfer between the hardware and the game engine.

Using Platform Designer the following features were implemented: a button is used to jump; the player tilts left/right to navigate in the desired direction; the FPGA is shaken vertically to implement a dash feature which gives the player a short burst of speed; score displayed on the 7-segment hex display.

In Platform Designer the onboard NIOS II processor was connected to the button, SPI accelerometer, JTAG UART and 7-segment display on the FPGA. This soft-core could then be programmed to process take data from the SPI accelerometer and whether the button has been pressed to pass that to the JTAG UART, and then receive the score from the JTAG UART to place that on the 7-segment display. The integrated SPI accelerometer was used to measure left-right tilt as well as vertical acceleration.

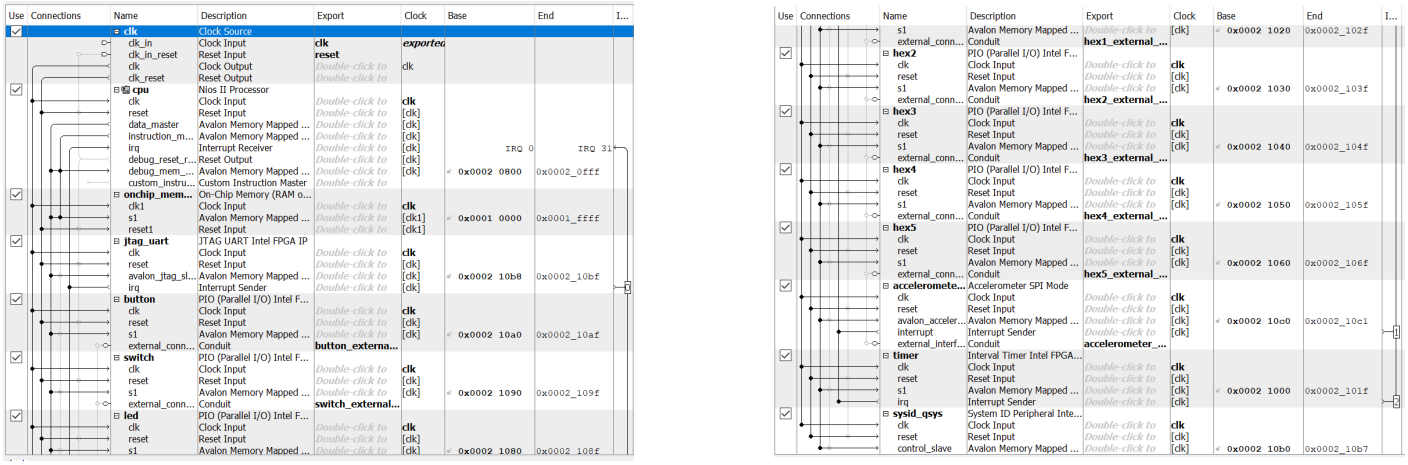

Figure 2: Hardware Components of FPGA as shown in Platform Designer

2.1.1 Firmware

Accessing the necessary hardware on the DE10-Lite requires flashing the FPGA with a Programmer Object File (.pof). This programming is persistent even after a power cycle, as opposed to using a SRAM Object File (.sof) which is volatile - this removes the requirement of having to flash the hardware every single time.

The POF is generated from a QSYS file that defines the I/O and hardware peripherals to be used (e.g. button, accelerometer, etc).

2.1.2 Software

The DE10-Lite can be programmed with C code, with extra libraries to provide seamless support for the DE10-Lite. The program uploaded would:

- Obtain readings from the accelerometer and buttons.
- Parse raw hexadecimal values from the accelerometer into decimal format using bit manipulations.
- Generate values for speed, direction, and booleans for jumping/dashing status.
- Send obtained values over UART/JTAG in a formatted string format to be easily parsed to a python program.

A circular buffer was used in determining whether the FPGA was shaken vertically. Initially the buffer is written to sequentially by iterating through it. Once the buffer is full, the index points back to the beginning of the buffer to overwrite the stored value and to continue re-iterating through the buffer. This avoids the need to constantly shift values in the buffer before a read/write, making efficient use of resources, which are typically relatively limited on embedded systems when compared to full-fledged CPUs.

Both the direction of tilt and the magnitude of tilt were measured. Accelerometer readings are in the form of 8-bit integers sign-extended to 32 bits. Bit masks were used to read the first bit of the value to determine its sign. A negative sign indicated a right-leaning tilt, and vice versa. This was implemented on the FPGA as opposed to the Python interface to the game. C is a compiled language, making it faster than interpreted languages such as Python. Speed is important, as we want the game to react near-instantaneously to the user in order to be playable - and subsequently fun.

The values were conveyed over UART character strings in a human-readable format, making parsing by the Python side simple. Values were sent via the alt-printf() function. Since this function could not parse the hexadecimal values into integer format out-of-the-box, this had to be implemented manually using bit manipulations. The integer values sent were the integer representation of bits 4 to 8 of the 32-bit integer, thus ranging between 0 and 15. For example, a value of 0x67 would be parsed as 6, and a value of 0xB8 would be parsed as 11.

The parsed values must then be transmitted to the game in a format it can interpret. To achieve this, the `intel_jtag_uart` library is used. It reads values from the FPGA using the .read() function and sends them to the game via UDP. The game's UDP server then interprets these values in player.gd.

Middleware was required between the FPGA and the Godot game engine as we had initially tried Godot objects to read UART which where incompatible with the intel FPGA we have. Therefore, we decided to implement python middleware found in the GitHub as udp.py which needs to deal with sending the score to the FPGA via UART and also reading what processing the Nios II processor is outputting.

Initially, TCP was used for communication between the FPGA and the game via local ports, but it proved too slow due to the handshake process, which caused delays in establishing connections and was not sufficiently responsive for the game. To improve responsiveness, two UDP servers were implemented instead, as UDP allows for instant connections and significantly faster communication, making the game much more responsive.

One UDP server is used for sending the movements from the FPGA's UART to the game which can then be interpreted by the game to translate to movements which are left, right, jump, dash via using the .read() from the intel_jtag_uart python library. This creates a port on the local host of 127.0.0.1:PORT at which the udp.py will send the data read from the JTAG UART which the game can then receive and decode for motion. The main issue with this method is that packets arrive in a discrete format, while character control in time is continuous. This requires smoothing techniques to ensure fluid movement. To achieve this, linear interpolation (`lerp`) is used. Instead of instantly jumping to the next speed, `lerp` calculates a weighted average between the current and next speed to become the actual next speed, resulting in smoother movement.

Another UDP client/server is set up to convey the score from the game (in this case, the client) to the Python script (the server) which sets up a port on the local host in the format again of 127.0.0.1:DIFFERENT=-PORT which the game can send the score to. The `.write()` function is used to send the score over UART. alt-getchar() is used to record the incoming UART data to a variable, and each digit is parsed into the required hexadecimal number, to be displayed on the 7-segment display.

UDP is a suitable choice as the port being targeted is local and so no packets will be lost in the communication, there is no handshake as in TCP removing that delay in gameplay experience and also it is faster than TCP allowing for a smoother gameplay experience for the user.

2.2 Game

The game was made using the Godot Game Engine. Godot was chosen over alternatives like Pygame due to the fact that it has an inbuilt realistic physics engine which would make the game feel more natural. In addition to since Godot is optimised for game engines checking for collisions with the platforms doesn't slow down the game noticeably whereas in Pygame having to check through so many bodies for collisions could hurt performance significantly.

Godot instantiates scenes that are a collection of nodes that allow you to build your game. Scripts which are used in various scenes can be written to implement node behaviour and interaction. Our design process started with adding a player and platform node allocating sprites to be displayed to the user and collision layers for interaction with each other. Player movement was added using a script mapping movement to keyboard input. The player also has a dash which temporarily boosts their horizontal velocity which can be used every 0.5 seconds. The game has three types of platforms that you can jump on these being regular platforms, platforms that break after one second after you land on them and moving platforms which are implemented using scripts. Following on from this, we added a kill zone that gradually moves up the screen to remove the players and platforms from the scene when they enter the zone and a coin that lets the player win when they touch it. This meant weha to add scenes for a win and death screen that the game takes you to when you win or lose.

To integrate multiplayer we had to add a start screen where you select your player. Once a player is selected they are taken to a wait screen where they wait for the other player to be ready to start the game. This required AWS integration to send signals to Godot to indicate when the players are ready (see: 2.3 AWS). The way multiplayer works is that on your screen you are player 1 and your position is sent to AWS where it is sent to the other client to render you on their screen.

The game was designed so that it is deterministic which allows the game states to be synced more easily. Since it is deterministic, while the game is being played the only thing that needs to be synced is the player position and the game will handle syncing the rest.

2.3 AWS

Our game is multiplayer and therefore a way of communication must be implemented in order to synchronise gameplay between players - we use TCP, with one server and 2 clients.

The server is a Python script running on an AWS EC2 instance. When started, it uses an algorithm to fill three arrays with randomly generated coordinates for normal, breaking, and moving platforms. These platform positions are then stored in a dictionary and sent to a DynamoDB database for persistence.

Once the platform data is prepared, the server waits for two clients to connect. The clients connect due to the IP address and port of the EC2 instance being run, being specified on the client script. Once both clients have joined, the server sends the platform data in JSON format to each client. We used JSON because it is easy for the Python and Godot scripts to parse and generate.

The client `global.gd` script, for each game instance, parses the received JSON data and uses it to instantiate the specified platforms - meaning both clients' screens display identical platforms. This ensures a synchronized game environment.

During gameplay, each player must be able to see the other if they are within the visible screen area. To achieve this, each client continuously sends its player ID and position to the server. The server then relays each player's position to the other client, ensuring real-time synchronization. The client uses this data to render the other player's character on its screen. A player doesn't join the game, unless the server receives 2 valid player ID's - this is when it will send a flag back to both clients to start the game. A client only sends a valid player ID once they choose a player (via a button click).

It is crucial for the game to completely synchronise real-time player position between the 2 clients as well as transfer all platform data reliably. Communication using TCP for the game server guarantees that any data transfer, arrives intact and in the correct order. Also, TCP has built-in error checking and retransmission - this eliminates the need to implement additional logic to handle lost packets, whereas UDP requires this. This is why we made the decision of TCP over UDP, despite UDP's faster transmission and lower latency.

3 Testing

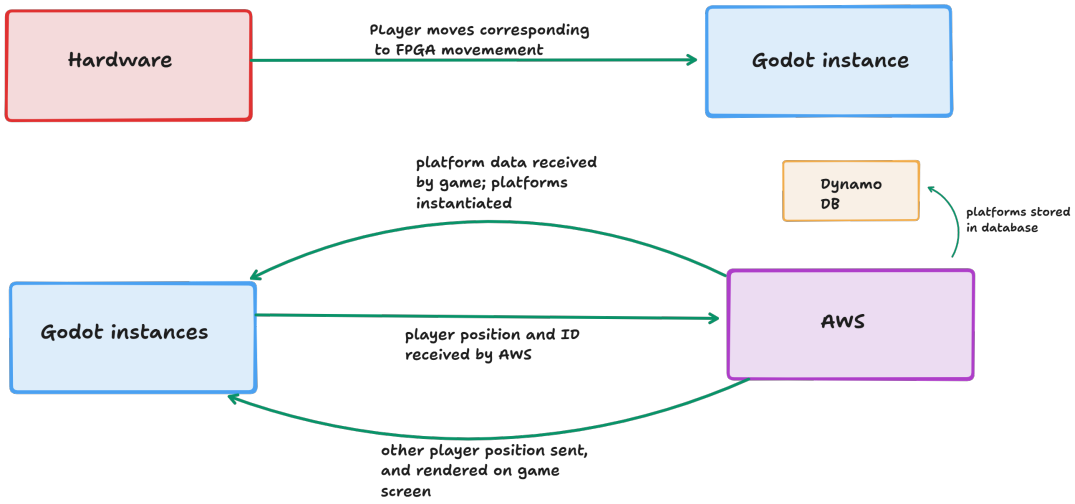

Figure 3: Testing flow chart

3.1 FPGA

Testing was an iterative process, by which the game was repeatedly played to see if the controls and sensitivities were easy to use.

The DASH-THRESHOLD variable, used in recognising a dash movement from the processing of accelerometer readings, was an example of iterative fine-tuning. Through trial and error, we eventually settled on a value that was neither too sensitive to small movements, nor too unreactive to moderately fast movements.

A bug that had been caught from early testing was the delay in flipping direction from left to right. This was initially solved by deleting the usleep(10000) command called at the end of the code's while loop, that would have otherwise introduced a delay of 10ms with each loop. However, this was later proven to cause the controller to crash after an unpredictable amount of time. Thus the delay was reinstated to a smaller amount, 5000 microseconds, and both problems were therefore eliminated.

Transitioning from a three-speed system (slow, medium, fast) to one with multiple discrete values (0 to 15) significantly smoothed and improved gameplay. This allowed us to test an acceleration factor based on the FPGA's tilt, enhancing the experience. However, in the end we opted for a method where the sprite's speed is directly proportional to the tilt angle.

During testing we had also initially implemented TCP to connect the FPGA to the game and as expected there were multiple issues with latency and time wasted for a TCP handshake. Therefore, UDP was the far superior choice in this case and offered a lower level solution to alternatives like using a Flask server with a higher level of abstraction with the HTTP whilst maintaining all required functionalities. Also, due to the speed of the UDP and JTAG UART there were issues with the Godot being flooded with the movement data, in order to deal with this a small delay was added into the python script which was tuned using trial and error to achieve minimal delay whilst keeping the game working settling with a 50ms delay which was practically unnoticeable and prevented flooding.

A key performance metric for the hardware is that the player movement should have the minimum latency possible and enable smooth controlling of the player, and this was made possible by maximising the frequency at which the NIOS II processes the SPI accelerometer data and by maximising the frequency at which this is sent to the game. This performance metric was measured by considering the delay introduced at each stage of the process from the accelerometer: 5ms from the C code due to usleep, 20ms in the python to stop flooding the game engine and 1.05ms (average measured using custom code) in transit to the game. This gives a total delay of accelerometer to game delay of 26.05ms, giving a very smooth gaming experience.

3.2 Game

For testing the game, we first developed it to work with keyboard inputs, so that we can test it without using the FPGA as a controller. Then we tested the game repeatedly and adjusted values to ensure that the game was always possible as the spawning of platforms is done randomly within a set of constraints. Implementing a dash helped with traversing platforms that were horizontally further apart.

For the set of scenes used to hold players when they select their player number and then get held while waiting for the other player we tested all combinations of inputs to ensure the game functioned no matter the set of inputs even if the players both select player 1 as well as ensuring it is possible to play a single player version where the platforms are generated locally instead of being generated by the AWS server.

3.3 AWS

Two major problems we faced then overcame were data packet loss and incorrect player position mapping. Some position updates were getting lost due to the TCP buffer overflowing, causing jittery movements. We fixed this by adjusting the buffer size and ensuring messages were properly terminated. The other issue was incorrect player mapping, where players appeared in the wrong locations because the server sometimes sent updates with mismatched IDs. We resolved this by strictly associating player IDs before transmitting data. These fixes ensured smooth real-time synchronization and eliminated movement glitches.

We tested the RTT between the server and the game and got an RTT of around 6ms to send and receive player coordinates from the AWS server via TCP which is more than fast enough to get a player update once per frame if we are targeting 60 fps for which we need an update every 16.67ms.

4 Conclusion

In conclusion, we managed to design a fully functioning system that met all requirements for this project. The project idea of a multiplayer platformer game generated enough scope to showcase our skills and knowledge in each of the respective areas of the project. The multiplayer aspect of the game allowed us to integrate two nodes into the system, and the random platform allocation and player synchronisation gave the opportunity to make effective use of the server. However, some areas that could have been improved was adding functionality for more than 2 players and adding and endless game mode. On the whole, the project was enjoyable offering a challenging yet satisfying experience and greatly improved our skills in this area of information processing.